

PID : Packaging - Integration - Development

An integrated software development process
Tutorial Document

Robin Passama
CNRS Research Engineer
Robotic Department
LIRMM - UMR5506 - University Of Montpellier
passama@lirmm.fr

October 2013



Laboratoire
d'Informatique
de Robotique
et de Microélectronique
de Montpellier



Introduction

The main goal of this document is to provide a method that helps improving the overall quality of the code and applications produced by the robotics department of LIRMM. By improving the quality we aim at:

- simplifying the understanding and the modification of code produced by others.
- simplifying the reuse of software components we develop.
- pooling software developments between robotic teams.

To achieve this goal the Finding, Writing, Compiling, Testing, Version controlling and Documenting of the code are mandatory concerns. This document define in part 1, how the whole development process is organized, which tools are used and which concepts are handled. Then part 2 provides tutorials on many aspects, as well repository management as package definition.

Part I

Basic Concepts

The following subsections explain basic concepts that need to be understood to start working with the PID methodology. First of all, root concepts are:

- **package** : a package is the basic unit for code development, unit testing, source version controlling, deployment specification, documentation writing. A package provides some functional code that can be reused (libraries, executables, header files, scripts, etc.) and depends on other packages either for its compilation (static libraries, header files archives/folders) or for its runtime use (dynamic libraries, executables, script-like files). It can contains as well an entire huge software (e.g. operating system) as a very elementary piece of code (e.g. header files, a library, etc.). A package has two main different forms :
 - a source form. In this form, the package is a git repository that contains the package's source code. It is something "alive" that is continuously evolving along development process.
 - a binary form. In this form, the package is a collection of software artefacts (headers, configuration files, executables, libraries, documentation, etc.) deployed in a given place of a file system. It is something static (does not evolved) that is associated to a specific version of the package's source form. A binary package can be embedded into an archive in order to be easily retrieved and deployed on a file system.
- **workspace**: the workspace is the folder hierarchy in which the user develops source packages and deploys the binary packages he uses (third party or resulting from its own package deployment). The basic idea is to avoid to use system dependencies : every software artefact in the development process is then local and relative to the workspace, except of course artefacts bound to system dependencies.
- **package server** : a package server is a computer (accessible across the network) that hosts many packages (either source or binary). It centralizes the access to packages, and handles the rights of the users for each package. It is responsible of the global version control of packages. It can also provides tools to manage the development of packages it hosts (teams members, bugs and activities tracking/reports, wiki, version history and development branches, visualization, etc.).

The present document helps normalizing the development of packages inside workspace, and the way packages and workspaces are managed on development servers. As definition of concept and process is intrinsically bound to the concepts and process involved in the software tools used, the section first quickly present these tools. Then we define core concepts of the PID methodology based on those of the tools.

1 Tooling

For each development task in a package, a tool is used to achieve this task. To ease the portability of code, only **cross platforms** tools are used:

- **git** is used for the concurrent version control and patching of the code. It is also the tool used to deliver and retrieve the source code.
- **cmake** is used to manage the build process, but also deployment and test.
- **doxygen** is used to generate api documentation of source code.
- **latex** is the preferred language to write documents, since it allows to version the full content of file (raw text and structure of the document), as opposed to binary formats like Microsoft Word or Libre-Office that are not well handled by git.

Other tools used, like the compiler/linker (e.g. gcc) used with cmake, or the ssh agent used with git, or even development environment (e.g. Xcode, Eclipse) interfaced with cmake, git and doxygen, are supposed to be native (i.e. specific to the OS used) and so adapted by the user depending on the platform he uses and his own preferences.

2 Package

The package is the basic working unit for developers. A package :

- contains the functional source code.
- contains the tests source code.
- contains version control information files.

- contains the compilation files used to : build the source code and documentation, run tests, generate configuration files, install its resulting binary form in the workspace and generate an installable archive of its binary form.

The main idea is that a package is self-explanatory. It does not means it contains all code and artefacts it uses but it contains all information needed to satisfy its dependencies. In other words considering a given package, its installation process is done using this information and the information contained in its dependent packages.

One important concept when dealing with PID packages is the concept of **component** : it is a software artefact produced by a package. It may be either:

- a **library** : this is a component that can be reused by developers to build other components. We define three types of libraries : **shared** (runtime or load-time linked library), **static** (link time only linked library) and **header** (compile time only "linked" library).
- an **application** : this is a component for end-users or developers that define a runtime behaviour. We define three types of applications : application (made for end-user or developers when it is a runtime component), example (demonstrating how to use libraries) and test (used only internally to a package to test its other components).

2.1 Package Structure

A Package is generically structured according to the folder hierarchy defined below:

- the root folder of the package has the **name of the package**. This folder is basically a git repository which allows to manage concurrent work and version control on a package's content.
- the **.git** folder contains version control related information, automatically managed by the git tool.
- the root **.gitignore** file is used to exclude from version control some artefacts like temporary files.
- the **CMakeList.txt** file is used to describe how to build, install and test the whole package. It also contains meta-information on the package (authors and institutions, repository address, license, etc.).
- the **build** folder contains results from build process and contains two subdirectories: **release** and **debug**. Each of them contains the hierarchy of files and artefacts generated by the compilation process.
- the **src** folder contains sources files (.c/.cpp/.cc in C/C++) of libraries. Each subdirectory of **src** contains sources use to build one or more library and is itself hierarchically organized according to developers needs. Libraries provided by the package are defined by the CMakeList.txt file contained in the **src** folder.
- the **include** folder contains interface description files, typically exported (i.e. installed) headers files (.h, .hpp, .hh) in C/C++. Hierarchical organization in this directory is the same as in **src**. Non exported headers are let in the **src** folder, as they are not considered as a part of the interface of the package.
- the **apps** folder contains source files for applications, an application being an example of the usage of a library, a runtime component or a end-user software. Each subdirectory of **apps** contains sources for one or more built application and is hierarchically organized according to developers needs. Applications provided by the package are defined by the CMakeList.txt file contained in the **apps** folder.
- the **test** folder contains source files for test units. Each subdirectory of **test** contains sources for one or more test unit. Custom test programs

and running tests applied to the package are defined by the CMakeList.txt file contained in the **test** folder.

- the **share** folder contains user written documents and some specific files used by the build process. Its contains different basic subdirectories :
 - the **doxygen** folder contains a "default" Doxyfile.in file that is used by doxygen to generate API documentation. This file can be modified by the user to add additional information to the generated documentation. The folder can also contain additional resources (like images), hierarchically organized according to developers needs, used by doxygen to integrate additional information in the API documentation.
 - the **cmake** folder contains cmake scripts (notably find scripts) that the package uses to find external resources like libraries. This is the place used only for very specific resources for which no default cmake script is available.
 - the **config** folder contains configurations files used by libraries and applications/tests of the package.
 - the **doc** folder contains "hand-written" documents (e.g. README files, tutorials, design documents, etc.).

The **share** folder define a CMakeList.txt file that can be used to install resources of the **doc** and **config** folders.

- the **license.txt** file contains the license that applies to the source code produced in the package. This file is generated by the build process.

2.2 Package repository

Package repositories are GIT repositories, whose content is structured according to the previously defined pattern. GIT is used to version all text files used (C/C++ sources, cmake scripts, latex sources, etc.). Only source form of a package is a git repository not its binary forms.

2.2.1 Version Numbers as GIT Tags

A package is continuously evolving along time and git provide an internal version representation of this evolution. Nevertheless, this representation is so fine grained (each time a modification is committed) that it is not really understandable by persons not involved in the package development. That is why we need a version representation that can be either understandable by users and developers. These versions, called **release version** are defined according to a specific policy.

A **release version** can be viewed as a screen-shot of the git repository at a given time of package's life. It is associated to a number (called **release version number**) that is uniquely identifying the version. Technically, a version is represented as a **GIT tag** : a git tag memorizes a given state of the repository and is marked with a unique label that is used to retrieve this state. In our context the label of the git tag represents the release version number and the repository state pointed by the tag corresponds to the release version. The labelling of git tags representing release versions follows the pattern below:

- the release tags have the shape **vX.Y[.Z]**
- **X** is the major version number (starts with value 0). Change of major version number indicates that the code is no more completely backward compatible with previous versions. In other words, the interface of the package (some of the headers files it contains) has been modified in such a way that some functions have changed or disappeared, or the behaviour/meaning of existing functions completely changed. While X is 0 the version is considered as pre-released and so is not ready for use by third party developers.
- **Y** is the minor version number (starts with value 0). It indicates an improvement that is completely backward compatible with previous version with same major version number. In other words, the only a little change of existing behaviours occurred OR the interface of the package has been improved with new functionalities without breaking the way one uses the older functionalities.

- Z is the patch version (starts with value 0). It represents a simple bug fix or a security fix. A patch changes nothing to the interface (no new behaviour/functionality) and modify only in a minimal way the internal behaviour.

Each time a new version of a package is released, its version number must be incremented according to the previously defined rules and a corresponding git tag is created. Here are some examples:

- 0.1.0 is the first pre-released version of the package.
- 0.Y.Z. are early development pre-released version of the package.
- 1.0.0 is the first release of source code.
- 1.2.0 is a release of source code backward compatible with version 1.0.0.
- 1.2.5 is a release of source code of version 1.2.0 with 5 bug/security fixes.
- 2.0.0 is a release that is no more backward compatible with 1.X.Y versions.

2.2.2 Development process with GIT branches

GIT branches are used to organize the development workflow by registering "increments" made in the repository. Increments are modifications of the repository content, either source code, documentation, etc. A git repository can have many branches representing parallel development work. Most of time developers create branches to isolate the work they do on a specific concern as regard of the software development. This work is keep isolated from the work made on other concerns until developers think this is the good time to integrate (or discard) them. From time to time GIT branches are created, deleted, and merged. Merging consists in registering modifications made in two or more branches into a single branch.

As GIT branches can be used to represent any type of concern their understanding can quickly become a real problem. That is why their usage is constrained according to a predefined pattern inspired from successful branching models. This pattern defines what is the meaning of branches, what they are use for and the way they are created and merged :

Main branches (see in figure 1) have infinite lifetime and must always be usable : their last commit must point to a state in which the package is compilable and executable with unit tests successful.

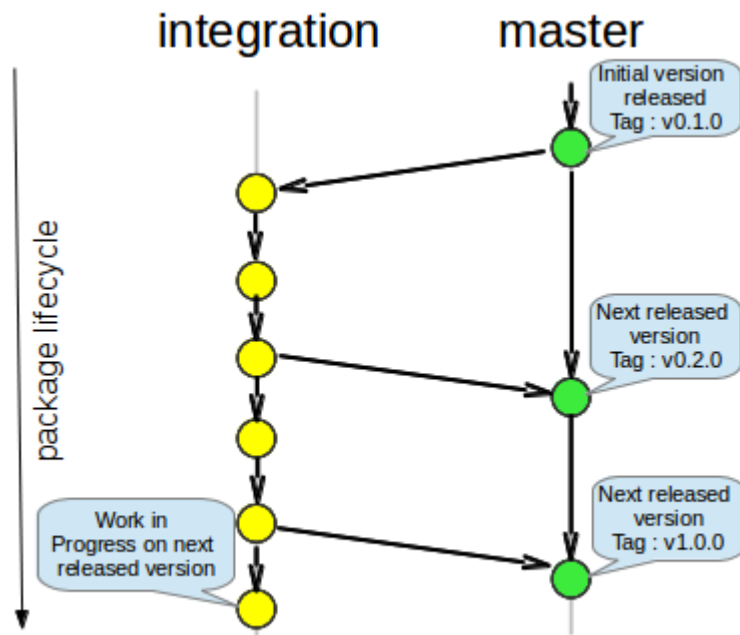


Figure 1: Permanent branches of a package

- The **master** branch contains the source code that always reflects a production-ready state. This branch is used to **tag** the released stable source code with version numbers but also to tag important intermediate states of the repository that reflect the development made for demonstrations and publications.
- The **integration** branch contains the detailed history of all the modification that have been realized on the repository. The source code of HEAD (pointer on the current state of the repository) always reflects a state with the latest delivered development changes for the next release. This is where any automatic nightly builds are built from, if any.
- When the source code in the **integration** branch reaches a stable point and is ready to be released, all of the changes should be merged back into **master** somehow and then tagged with an adequate release number.

Supporting branches are temporary branches, used to aid parallel development between team members, ease tracking of features and to assist in quickly fixing live production problems.

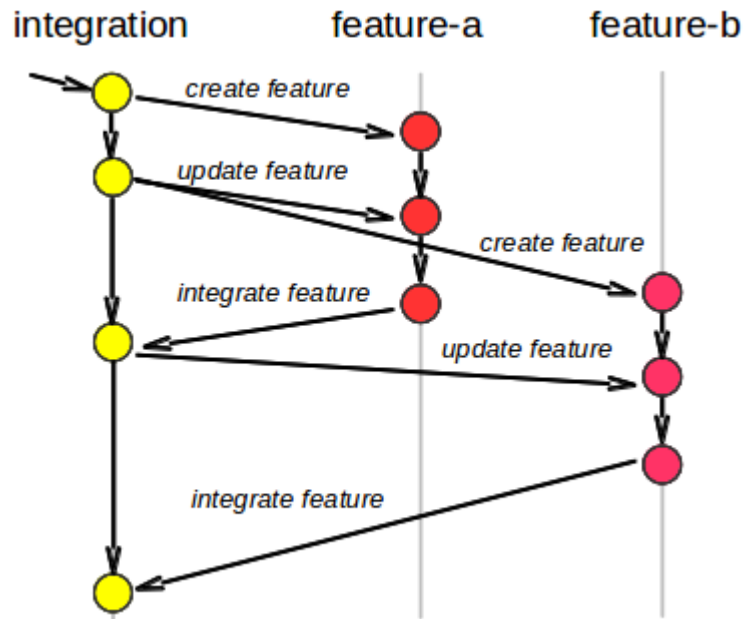


Figure 2: Relation between feature and integration branches

- **Features** branches (see figure 2) are used to develop new features /address new topics for the upcoming or a distant future release. Each feature branch must branch from the **integration** branch. A feature branch exists as long as the feature is in development, but will be merged back into develop (to definitely add the new feature to the upcoming release) or discarded (in case of a disappointing experiment). During its lifetime a feature should be updated from time to time with the modifications contained in the **integration** branch (issued for instance from the integration of other features). Doing so, the final merge of the feature will be more easy as the state of the feature will be not too far, in terms of importance of modifications, from the state of the **integration** branch.
- **Hotfixes** branches (see figure 3) arise from the necessity to act immediately upon an undesired state of a released version. When a critical bug in a released version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the **master** branch that marks the production version. Hotfix branches are used to allow team members to continue their work (on the **integration** and **feature** branches), while another person is preparing a quick bug/security

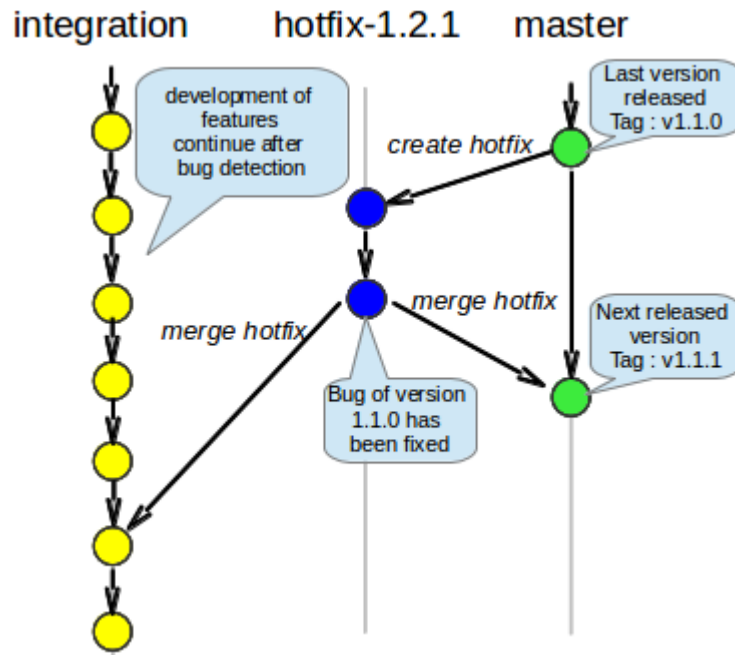


Figure 3: Relation between hotfix and integration/master branches

fix. When bugs are solved, the bug fix must be merged back into both **master** and **integration** branches. When merged the **master** branch is tagged with a new *patch version number*.

Naming Conventions:

- a **feature** branch name starts with "feature-" and ends with the name of the feature (given by developer).
example : feature-newfunctionality
- a **hotfix** branch name starts with "hotfix-" and ends with the new *patch version* of the released version.
example : hotfix-1.2.1

2.2.3 Collaborative working with GIT repositories

Now that the way the package is structured and is evolving the last part of this section consist in defining how people involved in the package's life cycle work together. The first concern is to define the **privileges** owned by these persons as regard of the repository usage. To do so we first define the three roles associated to each package:

- **users** are people using the package but not involved in its development.
- **developers** are users involved in the development of a package.
- **administrators** are developers with additional privileges, as they are considered as responsible of the package by users of the package.

Privileges associated to each role are managed inside the **package server** that hosts the package:

- each package repository is associated to three *groups*, each one representing a role previously defined. For example, for a package "a-pack", there are groups "a-pack-users", "a-pack-developers" and "a-pack-administrators". Each group provides specific privileges on the package repository.
- users registered inside the **package server** may be affected to one of these *groups*. Doing so, these users obtain corresponding privileges on the package repository.
- the repository can be set "public" so that anyone is considered as a user. In this case the users group may be not useful and can be let undefined.

The basic scheme for collaborative working between package developers is presented in figure 4. Given a package, this package has an **official GIT repository** that is deployed on a given **package server**. This repository is *official* because it centralizes information of the package and is public, which means that all concerned people (team, laboratory or more generally anyone registered in the server) can access to it. The access in itself is restricted with respects to roles:

- registered **administrators** have read/write access on the official package.
- all other registered **users** have read access.
- unregistered people may have read access (i.e. open source repository).

During development, one or more **private repositories** of the package can be created by *forking* the official repository. Depending on the official repository either only registered administrators (private package), developers (protected package), users (public package), or anyone (open package) can *fork* the official repository. Then the creator in addition of the administrators

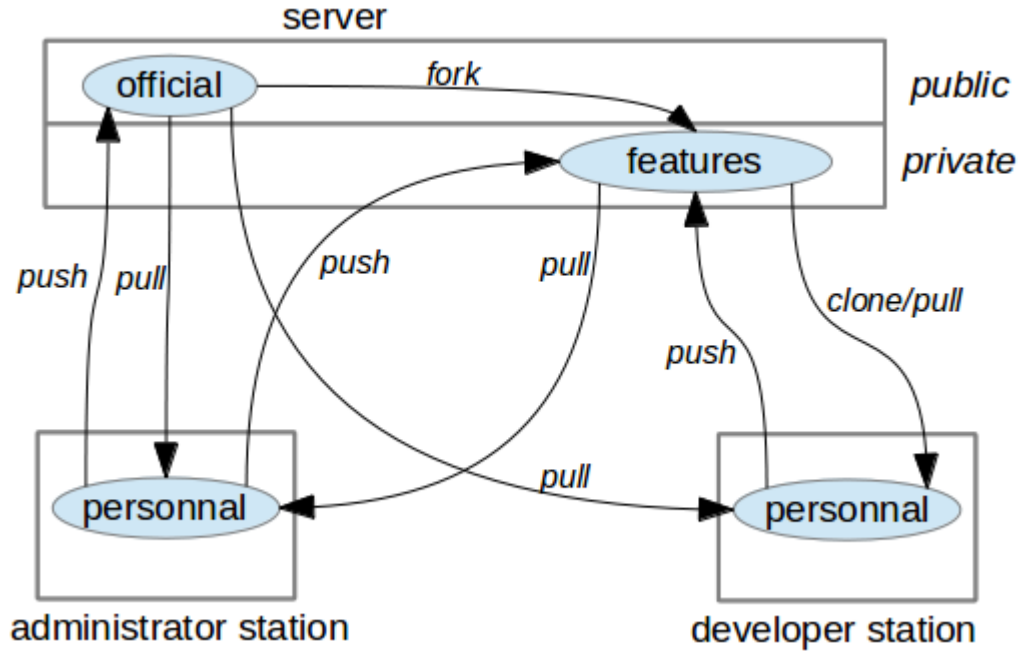


Figure 4: Collaboration between developers and repository

of the official, become **administrators** of the private repository. Then they can register new **users** and **developers** for this repository, with following privileges:

- registered **administrators** and **developers** have read/write access.
- registered **users** have read access.
- unregistered people have no access.

These repositories help structuring the development process by grouping developers work around a common repository while not immediately impacting the official one. This lets the time to the **administrators** to check if everything is OK and that modifications can be merged back into the official repository. The main purpose when using **private repositories**, is to separate the work on the same package made by different groups. For instance, a PhD student working on its demos on one side and a team with external people working in a common research project on the other side.

Administrators of the official package are the only ones that can update the **official repository** with modifications made on private repositories. That is why they are responsible of *version releasing* (see previous section):

they check the work done by developers and decides if the code is stable enough to release it. Furthermore, if there are many private repositories of the same package they act as brokers of changes made in separate pools of developers.

3 Workspace

The workspace is the place where developers work with packages (either those they develop or those they simply use). Technically, workspaces are local folder hierarchies in any station (server/demo/developers computers) that works with PID packages. A workspace is the place where packages repositories are created, developed and installed, and referenced given a pre-determined pattern. As compared to classical "do as you wish" installation, the "workspace way of doing" has several advantages:

- there is a reference path in the deploy process in such a way that classifying and finding software artefacts is made easy.
- there is no need to use system dependencies intensively for developed packages. As far as possible, every dependency should be satisfied by workspace content, except for system dependencies that cannot be packaged into workspace.
- developers can handle many version of a package in the same build/execution environment, without risking to break OS dependencies. Furthermore they can use multiple versions of the same package in the same time.

3.1 Workspace organization

3.1.1 Overview

A workspace is a folder (preferably named "workspace") with the following structure:

- the **.git** folder contains version control related information, managed by the git tool.
- the **install** folder contains packages' binaries installed by the user.
- the **packages** folder contains all the packages developed by an author. Each sub-folder of **packages** is a local repository of a given package as presented previously.
- the **external** folder contains all the external packages used by the installed package (source or binary). Each subdirectory corresponds to an external package, whose content is in turn completely undefined.
- the **.gitignore** file is used to exclude **install**, **packages** and **external** folders content from version control. This is mandatory since these folders content is supposed to be **purely local a a user workstation**.

- the **share** folder contains important files used to manage packages. The **docs** sub-folder contains documentation including the present document ; the **cmake** folder contains all scripts and data required to describe, build, install, deploy and configure packages: the **find** sub-folder contains cmake find script for commonly used external packages ; the **system** sub-folder contains generic scripts ; **patterns** contains cmake pattern files used ; **references** contains cmake script files with meta-information about available packages ; **licenses** contains cmake script files containing available license description;

3.1.2 Installing binary packages in the workspace

The install folder contains installed binary version(s) of any package used by a user. Each of its direct sub directories is a folder representing a given installed package, that itself contains :

- as many folders as there are concurrent binary versions of the package installed. The name of the version folder reflects the installed package version it contains (e.g. : 1.3.5 ; 1.2.0, etc.). Version folder can be also local (correspond to an install from a source package currently developed in the workspace), in such a case they name is of the form : own-0.2.5, own-1.0.0, etc.)
- an **installers** folder that contains all installable binary archives with each archive that corresponds to a specific version, for a specific system (linux, mac), in a specific mode (release, debug).

Each version folder is organized according to the following structure:

- the **bin** folder contains all executables provided by the package, except tests, in both debug and release modes. In other words it contains the result of the compilation of its corresponding package repository **include** folder.
- the **include** folder contains the exported interfaces of libraries. Basically, its direct sub-folders are libraries' root folder hierarchically organized the same way as in the package repository **include** folder.
- the **lib** folder contains libraries provided by the of the package in both debug and release modes. In other words it contains the result of the compilation of its corresponding package repository **src** folder.
- the **share** folder contains documents and scripts that are useful to the package users: the Use<Package><Version>.cmake file is a specific

script file used to identify elements and dependencies of the binary package ; its **doc** sub-folder contains API documentation generated by **doxygen** ; the **cmake** sub-folder contains cmake scripts files that are required to use the package, like find scripts used to configure external packages ; the **config** sub-folder contains installed element contained in the corresponding config folder of the package repository **src** folder.

- the **license.txt** file describes the license that applies to the software. This is a copy of the license file in package repository.

3.2 Workspace repository

The **official workspace** is a git repository that can be modified only by **administrators** of the server. It contains cmake scripts used notably to:

- reference available packages (repositories and binary archives). Each time a new package is created it is referenced into the local workspace (by an administrator) and then changes in the official workspace are committed so that anyone can know the existence of this new package and can retrieve its hosting server. This does not mean that this person can use binary version or repository of this package, since this is bound to his access rights to the servers (at least one can know which person to call to get the rights).
- provide available licenses description. These descriptions will be used in packages description.

The **official workspace** can be forked into **private workspaces**, exactly the same way as an **official package** (see figure 4), to provide a common workspace for a team developing one or more packages at the same time. Once created new **users** and **developers** can be added to the private workspace repository. Private workspaces will be updated (by **developers** and **administrators**) with new references to packages, new licenses and new find scripts, while new packages are implemented. Then **official workspace** administrators can update its content, at any time, with the modifications made inside the private repository.

Part II

Tutorials

This part of the document provides many tutorials

4 Work-flow management with git

Now that all basic concepts and principles have been described, this last section shows how to use them during the development process.

4.1 Installing Workspace and Packages

The first phase when starting development consists in installing a workspace on the local station of a developer or administrator, and configuring it adequately.

4.1.1 Getting Last Workspace Version

This phase simply consists in cloning the workspace repository. Any member of the robotic department can clone this repository, but only administrators can modify it.

```
git clone <global account>@<official server>:workspace.git
```

When done the developer can start his work. The workspace is empty, which means it only contains available licenses description in **licenses** folder, references to available packages in **references** folder, generic package structure in **patterns** folder, **CMakeModules** folder with existing cmake scripts and a complete **categories** folder with links that target nothing. Its **packages** and **frameworks** are empty.

We have to notice that only **administrators** have direct write access to the official servers and developers may or not have read access. Consequently, developers cannot modify the workspace repository.

4.1.2 Starting Development

Then the **developer** has two choices:

- either he starts the development of a new package,
- or he starts working on an existing package.

When a new package is created:

1. an **administrator** creates an empty GIT repository and optionally an associated (sub)project (e.g. a *redmine* project) on a given server (team or laboratory or open).
2. the **administrator** initializes the repository. On his local station he does:

```
cd <workspace path>/packages/  
git clone <global account>@<official server address>:<new package>.git  
cd <new package>  
cp -R ../../../../patterns/package/ .  
git tag -a v0.0.0 -m "creation of package"  
git checkout -b develop master  
git commit -a -m "initialization of package done"  
git push origin master  
git push origin develop
```
3. the **administrator** creates a private account on the same or another server (or uses an existing one) and he clones the new repository in this account:

```
git clone <global account>@<official server address>:<new package>.git  
cd <new package>
```
4. the **administrator** creates **feature branches** on private repository, by doing:

```
git checkout -b feature-<feature name> develop
```
5. the **administrator** registers the new developer (and any other person that may be involved in the development of this package) with ssh, so that he can access the git repository.
6. now **developers** can update/work with the private repository as they wish. To do this on their local working station they do :

```
cd <workspace path>/packages/  
git clone <private account>@<any server address>:<new package>.git
```

When done the development can start inside a package that is generic: it contains only the folder hierarchy and pattern files (e.g. doxyfile, CMake-List.txt, package.manifest, .gitignore) that matches the package structure defined in section 1.

When the developer start working on an existing package: He either works on existing features or on new features. In the first case the **administrator** simply has to register the **developer** on the existing private server (step 5-6). In the second case, the administrator has to choose:

- if a new private repository has to be created, for instance, in order to isolate the work made on new branches (steps 3 to 6).
- or if the developer uses an existing private repository with new branches (steps 4 to 6).

Whatever the solution chosen, the development starts in a package that already contains some specific software artefacts and has "generic files" already configured. The developer may then need to install new existing packages.

4.1.3 Installing Required Packages in the Workspace

When starting development with an existing package, this later can have one or more dependencies with another packages and with system. System dependencies must be fulfilled using classical install process provided by the OS. They can be found when compiling the package for the first time with **cmake**. Package dependencies are managed in another way:

1. to know which package are required, the developer has to look at the **package.manifest** file. The **dependency** markups list the required packages.
2. then the developer has to download and install these packages in its local workspace. To do so there are some alternatives:
 - the developer has a **read access** to the official repository of the package (it is listed in the corresponding **dependency** markup). In this case he can simply do:

```
cd <workspace path>/packages/  
git clone <global account>@<official server>:<other package>.git  
cd <other package>  
git checkout tags/v<version compatible with requirements>  
cd build  
cmake ..  
make  
make install
```
 - The developer can take a look at the **<package name>.manifest** file in the **references** folder of the workspace to know if there are

```

installers available and if they match the required version. In this
case, the developer uses the OS installer system and target the
required package framework. For instance, for a package that is a
simple archive :
cd <workspace path>/frameworks/
ftp ftp://ftp.gforge.fr/<other package>-<version>.tgz
tar -xzvf <other package>-<version>.tar.gz

```

3. When the last operation has been repeated for all packages, the frameworks of all packages contain adequate versions. Direct dependencies are resolved, but these required packages can in turn require other packages and OS dependencies. The current procedure has to be repeated recursively for each of these indirectly required packages.

Once all the dependencies are satisfied, the development can truly start.

4.2 Collaborative Software Development

All **developers** only work around **private repositories** of packages, while **administrators** are responsible of the update of **official repositories** according to the changes done in private repositories. Nearly all developments in **private repositories** are made in **feature branches**. The **develop branch** is used to merge and to solve conflict between features.

4.2.1 Developing a Feature

Common feature branches are already created, so the only thing to do for developers is to navigate between available feature branches and add content in branches :

- Each time a developer wants to start making modification of a given feature (for instance at the beginning of the day):

```
git checkout feature-<feature name>
git pull origin feature-<feature name>
=> solving potential conflicts
```
- During development of a feature, developers need to frequently "save" their work locally:

```
git add <modified files>
git commit -m "<telling modifications>"
```
- Each time a developer wants to stop making modification of a given feature (for instance at the end of the day) or when important modification

```
have          been          finished          and          committed:
git pull origin feature-<feature name>
=> solving potential conflicts
git push origin feature-<feature name>
```

During development on a feature branch, a developer may need to test some ideas without disrupting the work of all people working on the feature. In that case, he can work with a **new local branch** forked from feature branch. Modifications in this branch stay always local to the developer station. Furthermore modification should not be too big, these branches are used to test some ideas or to do bug fixes, not for developing big features (this is the role of feature branches) !

- Creating a local branch for testing an idea :
git checkout feature-<feature name>
git pull origin feature-<feature name>
=> solving potential conflicts and testing
git push origin feature-<feature name>
git checkout -b <idea name> feature-<feature name>
- During development of the idea, frequently save the work locally:
git add <all files modified>
git commit -m "<telling modifications>"
- When idea is OK and can be integrated to the feature:
git pull origin/feature-<feature name>
=> solving potential conflicts and testing
git rebase origin/feature-<feature name>
git checkout feature-<feature name>
git merge <idea>
git branch -d <idea>
git push origin feature-<feature name>
- otherwise if the idea is not good:
git checkout feature-<feature name>
git branch -d <idea>

The idea behind the use of the rebase command is to let the history of the feature development as linear as possible, in order to

4.2.2 Integrating Features

During development process, features have to be incrementally integrated as soon as their development is finished and they have been tested. The

integration can be done either by developers or administrators, but in all cases it requires every developers of the feature to be aware of this, to avoid big troubles.

1. Merging the feature in the development branch:

```
git checkout feature-<feature name>
git pull origin develop
=> solving conflicts and testing
git checkout develop
git merge --no-ff feature-<feature name>
=> solving conflicts and testing
```

2. Deleting the feature branch:

```
git push origin develop
git push origin --delete feature-<feature name>
```

3. Updating in local repository the branch that has been removed in remote repository (all users of the repository) :

```
git remote prune origin
git branch -d feature-<feature name>
```

Remarks:

- The development of features is made in parallel and they are merged indirectly in the **develop branch** at the very end, one at a time: features don't synchronize until merge. This let the possibility to developers to change some parts of the API/structure without immediately impacting the work made on others features.
- The best way is to first create an initial feature branch that puts in place the general basis of the package (API, basic class hierarchy, etc.). Then, when this feature has been merged in develop branch, parallel development into many feature branches can start.
- When merging, the resolution of conflicts **must be realized in feature branch** to avoid any problem in the develop branch while conflicts resolution take place.

4.3 Releasing a Package Version

Releasing package versions is the responsibility of administrators.

The first step is the update of the administrator station with modifications contained in private repository (named **private**):

1. Merging `develop` into `master`:

```
git checkout develop
git pull private develop
git checkout master
git merge --no-ff develop
doing test and solving conflicts, if any
Incrementing the version number in package.manifest file to get <new version number>.
git commit -a -m "Bumped version to <new version number>"
git tag -a v<new version number>
```
2. Updating private repository with new release version (to simply update the **package.manifest** file):

```
git checkout develop
Incrementing the version number in package.manifest file to get <new version number>.
git commit -a -m "Bumped version to <new version number>"
git push private develop
git push private master
```

The second step consists in updating the official repository (named `origin` for the administrator):

1. Getting branches from official repository:

```
git checkout develop
git pull origin develop
doing test and solving conflicts, if any
git checkout master
git pull origin master
doing test and solving conflicts, if any
```
2. Update the master and develop branches:

```
git push origin develop
git push origin master
```

4.4 Developing a Hotfix

Creating a hot fix is always made on demand of an administrator, but can be realized either by himself or a developer. The process is quite the same as for features, but we suppose that only one developer is involved in this task, that should be quick:

1. a package private repository is created or an existing one is used.

2. The administrator gets the last released version number from **package.manifest** file (e.g. 1.2.0) and increments the patch version number to get the <new patch version number> (e.g. 1.2.1).
3. Creating the hotfix branch in the private repository:
`git checkout -b hotfix-<new patch version number> master`
4. During the bug correction, committing locally:
`git checkout hotfix-<new patch version number>`
`git add <modified files>`
`git commit -m "<commit message>"`
5. Saving work in the private repository:
`git push origin hotfix-<new patch version number>`

When bug or security problems have been solved and tested, the development of the hotfix is finished. This is then the only time when the master branch can be modified in the private repository:

1. The responsible of the hotfix releases it:
`git checkout hotfix-<new patch version number>`
 changing version number in package.manifest file
`git commit -a -m "Bumped version to <new patch version number>"`
`git checkout master`
`git merge --no-ff hotfix-<new patch version number>`
`git tag -a v<new patch version number>`
`git push origin master`
`git checkout develop`
`git merge --no-ff hotfix-<new patch version number>`
 doing some conflicts resolution if necessary
`git push origin develop`
2. Then the hotfix branch is deleted:
`git branch -d hotfix-<new patch version number>`
`git push origin --delete hotfix-<new patch version number>`

Then the last step consists in releasing the patch version on the official server. This is the role of administrators that do that on their own station:

1. Updating master and develop branches:
`git pull private develop`
`git pull private master`
2. Checking that the hotfix solves the problem and that everything is OK.

3. Updating official repository branches:
`git push origin develop`
`git push origin master`

4.5 Updating workspace

Updating the workspace repository must be done:

- when a new package has been created and first released version has been published on package official server: the **package.manifest** file must be added and renamed to the **references** folder of the workspace.
- when a new version of an existing package has been release on a package official server: the corresponding **package.manifest** in the the **references** folder of the workspace must be updated.
- when a new license is used in a **package.manifest** file: the corresponding license file must be added to the **licenses** folder of the workspace.
- when categories defined in a **package.manifest** file have changed: the folders for new (sub)categories have to be created ; the link to the package framework have to be created or suppressed (when the package has been removed from a category).
- when a new package is released on official server and it targets some categories: links to that packages have to be added to target (sub)categories folders.

The update of the workspace is the work of the **administrators** only: they are responsible to guarantee the coherence of workspace repository structure and content.

4.5.1 Adding references to packages

When a package is added to the workspace official repository, on its local workstation an administrator of the package does:

1. Referencing the package in the workspace:
`cd <path to the package>`
`git pull origin master`
`git checkout master`
`cp package.manifest ../../references/<nameofpackage>.manifest`
`git commit -a -m "adding package <package name>"`

2. Looking package license in **package.manifest** and if the license does not exists in workspace create the corresponding license file:
`cd <path to the workspace>`
`cp patterns/license.license licenses/<licensename>.license`
 writing the new license file...
`git commit -a -m "adding license <license name>"`
3. Looking package categories in **package.manifest** file, and if categories do not exist, create them:
`cd <path to the workspace>/categories`
`mkdir <category path>`
`git commit -a -m "adding category <category name>"`
4. Adding a link to the package framework in each category folder defined in **package.manifest** file:
`cd <path to the workspace>/<path to category folder>`
`ln -s -t . <relative path to workspace root>/frameworks/<package name>`
`git commit -a -m "adding link to <package> in <category name>"`
5. If a package requires some system dependencies that the administrator think to be relevant for many developers, he can add the cmake script files used to configure these dependencies to the **CMakeModules** folder of the workspace.
`cd <path to the workspace>/CMakeModules`
`cp <path to the workspace>/packages/<name of package>/CMakeModules/<chosen dependency>.cmake .`
`git commit -a -m "adding cmake script for <dependency>"`
6. Updating the workspace official repository:
`cd <path to the workspace>`
`git pull origin master`
 solving potential conflicts
`git push origin master`

4.5.2 Removing references to packages

The process is quite the same as previously except that only the package reference file is suppressed from the workspace references folder, not the license files or the categories.

1. Removing the package reference:
`cd <path to the workspace>/references`

```
rm <package name>.manifest
git commit -a -m "removing package <package name>"
```

2. Updating the workspace official repository:
cd <path to the workspace>
git pull origin master
solving potential conflicts
git push origin master

4.5.3 Updating references to packages

The update of a package reference must be done **each time a new version of the package is released** in the official repository of the package. The process is the same as for adding the package reference for the first time, except for categories:

1. Looking package categories in the new version of the **package.manifest** file, and if categories do not exist, create them:
cd <path to the workspace>/categories
mkdir <category path>
git commit -a -m "adding category <category name>"
2. Adding a link to the package framework in each category folder defined in **package.manifest** file:
cd <path to the workspace>/<path to category folder>
ln -s -t . <relative path to workspace root>/frameworks/<package name>
git commit -a -m "adding link to <package> in <category name>"
3. Removing links in each category folder no more defined in **package.manifest** file:
rm <path to the workspace>/<path to old category folder>/<package name>
4. Updating the workspace official repository:
cd <path to the workspace>
git pull origin master
solving potential conflicts
git push origin master

5 Package development with CMake

When developing a package one need to handle information such as:

- *meta-information*: who is involved in its development ? what is its general purpose ? where to find this package on the network ? what is the license of the code ? etc.
- *build-related information*: what are the components provided by the package and how to compile/link them from source code ? How components are tested ? what is the version of the package ? what are its functional dependencies ?
- *functional information*: this is the source code of the components and other associated files like configuration files if any required.

The whole package development is organized and described with `cmake`, and the `CMakeList.txt` files used contain the whole *meta-information* and *build-related information* used by the package. Each package contains several `CMakeList.txt` files:

- the root `CMakeList.txt` file (direct leaf of the package source repository, see section 2.1) is used to define *meta-information* and *dependencies* of the package.
- the `CMakeList.txt` file contained in the **src** folder defines the **library components** (see section 2).
- the `CMakeList.txt` file contained in the **apps** folder defines the **application components** (see section 2).
- the `CMakeList.txt` file contained in the **test** folder defines the **test components** (see section 2) and running tests (using these components or others).
- the `CMakeList.txt` file contained in the **share** folder defines additional files to install, like configuration files used by libraries and/or applications (if any), documents, etc.

Each of these `CMakeList.txt` files must follow a predefined pattern. This pattern is mainly influenced by the use of PID specific CMake functions. The following subsections present examples on how to use these functions together with more classical CMake code in order to completely define a PID package.

5.1 Package's root CMakeList.txt file

5.1.1 General Meta-Information

Let's suppose we define a package with the name "the-testpack-b", its root CMakeList.txt file could look like this.

```
PROJECT(the-testpack-b)
CMAKE_MINIMUM_REQUIRED(VERSION 2.8.11)
set(WORKSPACE_DIR ${CMAKE_SOURCE_DIR}/../.. CACHE PATH "root of
the packages workspace directory")
list(APPEND CMAKE_MODULE_PATH ${WORKSPACE_DIR}/share/cmake/system)
# using generic scripts/modules of the workspace
include(Package_Definition)

declare_PID_Package(
    AUTHOR      Robin Passama
    INSTITUTION LIRMM
    YEAR        2013
    LICENSE      CeCILL
    ADDRESS      git@idh.lirmm.fr:perso/passama/the-testpack-b.git
    DESCRIPTION  test package B for PID
)

set_PID_Package_Version(1 1)

# adding some binary packages references
add_PID_Package_Reference(
    BINARY VERSION 0 1 0 SYSTEM linux
    URL
    http://lirmm.lirmm.fr/FileX/get?auto=1&k=rfYTf1gkpI5XtEpQWVA
    http://lirmm.lirmm.fr/FileX/get?auto=1&k=oMyg4JVeeKYWpqwEFxE
)

add_PID_Package_Reference(
    BINARY VERSION 1 1 SYSTEM linux
    URL
    http://lirmm.lirmm.fr/FileX/get?auto=1&k=DYt2j35Kw8ozOgHfoVA
    http://lirmm.lirmm.fr/FileX/get?auto=1&k=zEx02N4KWfzDWPTxi0
)
```

Exactly as in any CMake project, the CMake package is define with the PROJECT keyword. The project's name is **the name of the package**

root folder (and so the name of the repository). The remaining lines until the `declare_PID_Package` macro call must be let **unchanged** (initialization of the PID specific cmake scripting system).

Then comes the `declare_PID_Package` macro, which is mandatory. This macro defines general meta-information on the package, and should not change a lot during package life cycle:

- the main author (AUTHOR keyword) and its institution (INSTITUTION optional keyword), considered as the maintainer of the package. The main author is an **administrator** of the package (see section 2.2.3).
- the YEAR field helps defining the package's life cycle range. For instance one can input a field such as "2009-2013".
- the LICENSE field is used to specify the license that applies to the code. This license must be defined in the workspace (see section 3.1.1) in the form of a **cmake script license file**.
- the ADDRESS field is used to specify the address of the **official GIT repository** of the package (see section 2.2.3).
- the field DESCRIPTION must be filled with a short description of the package usage/utility.

Then the user fill other meta-informations that evolve during project life cycle:

- the `set_PID_Package_Version` function is used to set the currently developed version of the package. It take at least a MAJOR and MINOR numbers and optionally a PATCH number as arguments (default value for PATCH number is 0). The version number thus follows the same pattern as git **release versions**. Before a version is released with a git tag (see section 2.2.1) this version number must be set adequately so that git tag matches cmake package version. Generally, the best way to do is set the version number used in the CMakeList.txt with the number of the next version to release.
- the `add_PID_Package_Reference` function is used to register a downloadable binary version of the package. The VERSION keyword specify the version with MAJOR MINOR PATCH numbers. The SYSTEM

keyword specifies the target operating system for which the binaries have been built. The two addresses after the URL keyword specify where the binary package version can be downloaded either in release (first address) and debug (second address) modes.

5.1.2 Dependencies with other packages

The last part of the root CMakeList.txt is used to manage dependencies between the current package and other packages it depends on. It could look like this:

```
# finding used packages
find_package (Boost REQUIRED)
find_package (the-testpack-a 1.0 REQUIRED lib-b-sh lib-x)

# declare a dependency over Boost (by default boost root is the
# /usr system install dir)
if(Boost_DIR-NOTFOUND)
    set(BOOST_ROOT /usr)
endif()
declare_PID_Package_Dependency(PACKAGE Boost EXTERNAL
${BOOST_ROOT} VERSION "${Boost_MAJOR_VERSION}.
${Boost_MINOR_VERSION}.${Boost_SUBMINOR_VERSION}")

#declare a dependency over the-testpack-a PID package
declare_PID_Package_Dependency (
    PACKAGE the-testpack-a
    PID VERSION 1 0
    COMPONENTS lib-b-sh lib-x)

build_PID_Package()
```

The `find_package` function is a standard cmake function used to find and configure other packages. In the example, we search for a package named "the-testpack-a" (that is also a PID package) with version compatible with 1.0 (for PID packages compatibility as the same meaning as in section 2.2.1, otherwise the meaning is package dependent). The package is **REQUIRED** meaning that it must be found. We specifically require the **components** named "lib-b-sh" and "lib-x". As experienced CMake users may notice there is no difference in the usage of the `find_package` function as regard of its standard use.

Then PID development process imposes to declare dependencies of the current package. Indeed this is not because you try to find other package that you will use them, even if obviously this assumption will be right most of time. A dependency simply means that components of the package are using components from other packages (see section 2 to understand what is a component). For instance, the current package uses the package "the-testpack-a" with minimum version 1.0. To make this declaration possible PID provides the `declare_PID_Package_Dependency` function. This function can be used in two different ways:

- it is used to specify a dependency to an **external package** (using the keyword `EXTERNAL` after the package name). An external package is just a *reference* associated to a path that points to the root folder of a **NON PID** package. This reference will be used later when defining components. A version information can be appended but is not used. A good way to do is to install external package into the **external** folder of the workspace, when possible, so that they will be managed by PID in a easier way. External package are packages installed by hand by the user in his file system, either in system or add-hoc folders. For packages that are supposed to be part of the operating system, there is no need to specify the dependency since this dependency is satisfied "by default" by the system.
- it is used to specify a dependency to a PID package (using PID keyword). Then version and component requirement informations can be used exactly as in the example. Relationship between PID packages are stronger since their discovering/install configuration will be done automatically.

Finally, the `CMakeList.txt` file call the `build_PID_Package` function. This line is mandatory in order to allow the build of the package: compilation, installation, deployment, API doc generation, etc. Without this call, the CMake process would simply do nothing.

5.1.3 Dealing with conditional dependencies

The previous example is quite simple since it directly deals with **required** dependencies. Nevertheless, when using cmake one sometimes have to deal with **conditional dependencies**. Indeed, **conditional dependencies** allow to configure the build according to the OS requirements or to the configuration of user's station. This conditional dependencies are managed according to the following pattern (or a derivative from this solution):

```

find_package(the-testpack-a)
if(the-testpack-a_FOUND)
    set(USE_the-testpack-a TRUE CACHE INTERNAL "" FORCE)
    declare_PID_Package_Dependency (PACKAGE the-testpack-a
                                    PID VERSION 1 0 COMPONENTS lib-b-sh lib-x)
else()
    set(USE_the-testpack-a FALSE CACHE INTERNAL "" FORCE)
    find_package(the-testpack-x)
    if(the-testpack-x_FOUND)
        set(USE_the-testpack-x TRUE CACHE INTERNAL "" FORCE)
        declare_PID_Package_Dependency (PACKAGE the-testpack-x
                                        PID VERSION 1 3 COMPONENTS lib-any)
    else()
        set(USE_the-testpack-x FALSE CACHE INTERNAL "" FORCE)
        MESSAGE("alternative system dependency not satisfied.
        Install/select either the-testpack-a or the-testpack-x")
        return()
    endif()
endif()

```

Conditional dependencies is used to describe dependencies that are optional or to describe an alternative requirement when multiple different packages can be used for the same purpose. This later case is shown in the previous code. Defining such conditional dependencies is made using classical CMake mechanism (using `find_package` and `option/set` commands) is combination with the `declare_PID_Package_Dependency` function previously explained. In the previous example, there is an implicit priority in the usage of required packages: "the-testpack-a" will be used prior to "the-testpack-x" if it is found. The `CMakeList.txt` file can also use the `option` command to let the user select its preferred package. The only mandatory descriptive elements are:

- if a package is to be used, **whether it has been previously found or not**, then the `declare_PID_Package_Dependency` function **must be called** to tell to PID system which package (PID or external) is required.
- Each possibly used package must be bound to a CMake **variable that indicates if the package is really used or not**. For instance see the variable `USE_the-testpack-a`. These variables will be used later, when declaring components, to know how to configure component dependencies according to required packages.

5.2 Defining Library components

Once package dependencies have been defined, the package developers can then define components of the package and their relationship with these dependencies. Most common components are library components : they are used by developers to define reusable functionalities. All libraries are defined in the CMakeList.txt file contained in the **src** folder of the package repository.

PID defines three types of libraries, matching the three classical types available in C/C++. It provides a systematic way to define these libraries and automate their use, avoiding the user to know precisely how to deal with each type and their relative properties. Following subsections explain how to declare each of these types. These definitions rely on cmake functions provided by PID: `declare_PID_Component` and `declare_PID_Component_Dependency` (see appendix 7).

5.2.1 Header libraries

Header libraries are not compiled to produce a binary object. They are just made of a set of header files. This kind of library used is often used for template libraries definitions, for instance the Boost library. Such a library is never used at link time (never linked to another library or executable using a linker) but only at compile time (included header files). The definition of a header lib should look like:

```
declare_PID_Component(HEADER_LIB NAME lib-y DIRECTORY lib_y)
declare_PID_Component_Dependency(COMPONENT lib-y
                                EXPORT DEPEND lib-x PACKAGE the-testpack-a
                                EXPORTED_DEFINITIONS USE_EXTERNAL_LIB_X)
```

The first thing to do is to declare the header library by using the function `declare_PID_Component`:

- the `HEADER_LIB` keyword is used to declare a header library.
- the `NAME` keyword is used to define the identifier of the component in PID, whose unicity must be preserved. In the example the name is "lib-y".
- the `DIRECTORY` keyword is used to say in which sub-directory of the **include** folder the header files are found, in the example the "lib_y" sub-folder. The direct consequence is that all headers of a given library **must be placed in a unique folder**.

Then, depending on the library content, some dependencies can be attached to the library, using `declare_PID_Component_Dependency`. Indeed a header library can depend on other libraries, either header, static or shared:

- the `COMPONENT` keyword is used to specify for which component a dependency is defined, in the example the previously defined "lib-y" header library.
- the `EXPORT` keyword specifies that the lib-y component export the required dependency. Exporting means that the reference to the required component is defined in the interface of the library. Since a header lib is only made of an interface, **it must export each of its dependencies**.
- the `DEPEND` and `PACKAGE` keywords are used to specify the dependency: here the lib-y header library depends on the component "lib-x" of the PID package "the-testpack-a". Declaring an external or system dependency or even an internal dependency is slightly, but follows the same logic, see appendix 7.
- the `EXPORTED_DEFINITIONS` is used to specify values of C preprocessor definitions that are exported by the library. In the example the exported definition is `USE_EXTERNAL_LIB_X`. exported definition are used by components that will use lib-y to configure its code adequately. Since a header lib is only made of an interface, all its definitions must be exported and it can have no internal definitions.

On interesting property of PID is to be able to declare different components from the same code. For instance:

```
declare_PID_Component(HEADER_LIB NAME lib-y-bis DIRECTORY lib_y
                     EXPORTED_DEFINITIONS NUSE_EXTERNAL_LIB_X)
```

In this later example, a new component named "lib-y-bis" is created from the same source code contained in the `lib_y` folder. The differences with the previous component are that the "lib-y-bis" has no dependencies and it undefines (using `N` in front of the preprocessor definition) the definition `USE_EXTERNAL_LIB_X`. This is useful to declare many alternatives from the same code.

5.2.2 Static libraries

Static libraries are binary archives that define some functionalities. A static library is made of:

- a set of header files that define its interface (i.e. what is available for library users).
- a set of (compiled) binary objects that implement its behaviour.

Its interface is used at compile time (when its header are included) and its contained objects are linked to executables and shared libraries at link time, so they no more exist at run time. The definition of a static lib should look like:

```
declare_PID_Component(STATIC_LIB NAME lib-c-st DIRECTORY lib_c
                      INTERNAL DEFINITIONS NA_VERY_SPECIFIC_IMPLM)
declare_PID_Component_Dependency(COMPONENT lib-c-st
                                EXTERNAL Boost INCLUDE_DIRS <Boost>/include)
declare_PID_Component_Dependency(COMPONENT lib-c-st
                                EXPORT DEPEND lib-y-bis)
```

As for any component, the first thing to do is to declare by using the function `declare_PID_Component`:

- the `STATIC_LIB` keyword is used to declare a static library.
- the `NAME` keyword is used to define the identifier of the of the static library, "lib-c-st" in the example.
- the `DIRECTORY` keyword is used to say in which sub-directory of the **include** folder the header files of the static library are found, in the example the "lib_c" sub-folder. The **same folder name** is used to specify in which subdirectory of the **src** folder the source and non-public header files of the library are found. For a same library, this constraints the user to use same folder names between **include** and **src** directories.
- the `INTERNAL DEFINITIONS` is used to specify definitions that affect only the implementation (i.e. that is not used in any header file of the library). In the example the "lib-c-st" undefines the preprocessor definition `A_VERY_SPECIFIC_IMPLM`.

As readers can notice, the declaration is quite the same as for header libraries. Note also that static libraries can define exported definitions (as header libraries) for those which are used in their header files. The declaration of dependencies also follows the exact same pattern. In the example:

- "lib-c-st" static library is using an external package named "Boost". As boost is a pure header library it only needs to specify where to find its header files, using the `INCLUDE_DIRS` keyword. The include path specified is relative to the Boost package root folder (using the `<Boost>` specifier).
- "lib-c-st" is also using (specified with keyword `DEPEND`) another library "lib-y-bis" that is defined in the same package (since no `PACKAGE` keyword is used). It exports "lib-y-bis" meaning that its header files contain include directive over header files of the "lib-y-bis".

5.2.3 Shared libraries

Shared libraries are binary objects that define some functionalities. A shared library is made of:

- a set of header files that define its interface (i.e. what is available for library users).
- a binary object (.so on linux) that implements its behaviour. This

Its interface is used at compile time (when including its headers) and its binary object checked at link time and truly used at run time, either when the executable using it is loaded or when it explicitly load it at run time. The definition of a shared lib is more or less the same as for static libraries and should look like:

```
declare_PID_Component(SHARED_LIB NAME lib-c-sh DIRECTORY lib_c
                      INTERNAL DEFINITIONS A_VERY_SPECIFIC_IMPLM)
declare_PID_Component_Dependency(COMPONENT lib-c-sh
                                EXTERNAL Boost INCLUDE_DIRS <Boost>/include)
declare_PID_Component_Dependency(COMPONENT lib-c-sh
                                EXPORT DEPEND lib-y)
```

In the previous example, the function `declare_PID_Component` is used in a common way:

- the `SHARED_LIB` keyword is used to declare a shared library.
- the `NAME` keyword is used to declare "lib-c-sh".
- the `DIRECTORY` keyword is used to define **include** and **src** sub folders where to find code. In the example reader can notice that the shared library "lib-c-sh" is built from the same code as static library "lib-c-st".
- the `INTERNAL_DEFINITIONS` is used to specify to define the preprocessor definition `A_VERY_SPECIFIC_IMPL` contrarily to "lib-c-st".

This example shows how shared and static libraries can be built from the same source code and how developers can define alternative implementation for part of their code using preprocessor definitions. Their dependencies can also vary:

- "lib-c-sh" shared library is using the Boost external package the same way as "lib-c-st".
- "lib-c-sh" is using (specified with keyword `DEPEND`) the library "lib-y" (that is defined in the same package) instead of "lib-y-bis" used by "lib-c-st".

5.3 Defining Application components

5.3.1 standard applications

5.3.2 Example applications

5.3.3 Test applications

5.4 Defining Tests

5.5 Generating API documentation

5.6 Configuring the package

5.7 Installation of the package

ICI A REVOIR

First part defines the general settings of the project that should be quite the same for all packages:

- name of the cmake project is the name of the package.

- basic options for package users/developers are set in order to let the possibility to build examples and test units (i.e. non functional code).
- version number matches the current version number of the package. it **must be updated each time a new release version is tagged on master branch**.
- if a user defines the `USE_CUSTOM_DEPLOYMENT` variable, the deployment of the package in framework will be done in the **own** folder and not in the **version** folder.
- directories containing cmake script files used (e.g. to find os dependencies) are referenced in the `CMAKE_MODULE_PATH` variable.

The **apps** subdirectory contains a `CMakeList.txt` file that builds applications. The organization into subdirectories follows the same logic as for libraries:

```
## first subdirectory of apps: an executable ##
set(EXE_SRC_<exe name> "")
list(APPEND EXE_SRC_<exe name> <exe name>/<path to file>)
list(APPEND EXE_SRC_<exe name> <exe name>/<path to file>)
...
add_executable(<exe name> EXE_SRC_<exe name>)
target_link_libraries(<exe name> <lib name>_st)

if(<package>_WITH_EXAMPLES)
## second subdirectory of apps: an example executable ##
set(EXE_SRC_<exe name> "")
list(APPEND EXE_SRC_<exe name> <exe name>/<path to file>)
list(APPEND EXE_SRC_<exe name> <exe name>/<path to file>)
...
add_executable(<exe name> EXCLUDE_FROM_ALL EXE_SRC_<exe name>)
target_link_libraries(<exe name> <lib name>_sh)
...
## building example targets, if any ##
add_custom_target(examples COMMAND ${CMAKE_COMMAND} -E echo "Building
examples targets" DEPENDS <example 1> <example 2>...)
endif(<package>_WITH_EXAMPLES)
```

The main difference with libraries is the usage of the `add_executable` command to create executable and the use of a guard (`if(<package>_WITH_EXAMPLES)`) to avoid building examples applications if not required by the developer.

Finally, the **test** subdirectory contains a CMakeList.txt file that builds test units. The organization into subdirectories follows the same logic as for libraries and executable:

```
if(<package>_WITH_TESTS)
set(TEST_SRC_<test name> "")
## first subdirectory of test: a test unit ##
list(APPEND TEST_SRC_<test name> <test name>/<path to file>)
list(APPEND TEST_SRC_<test name> <test name>/<path to file>)
...
add_executable(<test name> EXCLUDE_FROM_ALL TEST_SRC_<test name>)
target_link_libraries(<test name> <lib name>_st)
...
## second subdirectory of test: a second unit ##
...
add_custom_target(testunits COMMAND ${CMAKE_COMMAND} -E echo "Building
test units" DEPENDS <test unit 1> <test unit 2>...)
...
```

The build process for tests is quite the same as for applications but all tests are conditionally built depending on developers choice, using a specific guard (if(<package>_WITH_TESTS)).

5.7.1 Defining tests

When the code has compiled and linked correctly a good practice is to test it to be sure it behaves properly or respects some backward compatibility constraints. To do so we use test units built during the build process or any other interesting test tool and directly integrate the testing in the CMakeFile.txt of the **test** folder using CTest tooling integrated in cmake:

```
...
enable_testing()

add_test(<test1 name> <test unit 1> arg1 arg2 ... )
add_test(<test2 name> <test unit 1> arg1 arg2 ... )
add_test(<test3 name> <test unit 2> arg1 arg2 ... )
...
endif(<package>_WITH_TESTS)
```

This will automatically generate a sequence of tests whose result is **PASSED** or **FAILED** once the user input the **make test** command. We can also log results of tests into a dashboard using the same CTest tool, but this discussion is beyond the topics of this document.

5.7.2 Building API documentation

When an API becomes available, it is always useful to have a clear way to consult the functionalities it provides. This API can be either provided by a runtime component, a library or a pure headers set. In all cases the api documentation has to be as close as possible to the source code, that is why the best way is to use an api doc generation tool like **doxygen** and to automate its use directly during the build process. We choose **doxygen** because it is a cross-platform/cross language tool very well suited for C/C++ code.

The generation takes place only if the **doxygen** tool is installed on the building host (`find_package(Doxygen)`) and only when release build mode is chose (`CMAKE_BUILD_TYPE MATCHES Release`). Furthermore we suppose that code documentation that can be exploited by doxygen is available in files contained in the **include** folder. The api generation is made in three steps :

1. generates a doxygen configuration file (**Doxyfile**) from a cmake doxy-file pattern (**Doxyfile.in**) that is configured according to some cmake variables. This file is used by doxygen to configure the api documentation generation process.
2. creates a specific target named **doc** that call doxygen with the previously generated configuration as argument.
3. once native build files have been generated by cmake, the documentation is produced with the command **make doc**.

All this process requires that the share folder contains a doxygen configuration file pattern named **Doxyfile.in**. This file is provided with default values with a package and should not be modified except by expert users. It looks like this:

```
...
DOXYFILE_ENCODING      = UTF-8

# The PROJECT_NAME tag is a single word
PROJECT_NAME           = "@DOXYFILE_PROJECT_NAME@"

# The PROJECT_NUMBER tag can be used to enter a version.
PROJECT_NUMBER         = "@DOXYFILE_PROJECT_VERSION@"

# The OUTPUT_DIRECTORY tag is used to specify the (relative or
```

```

# absolute) base path where the generated documentation will
# be put.
OUTPUT_DIRECTORY          = "@DOXYFILE_OUTPUT_DIR@"
...

# The IMAGE_PATH tag can be used to specify files or
# directories that contain image included in the documentation
IMAGE_PATH                = "@DOXYFILE_IMAGE_PATH@"
...

# If the GENERATE_HTML tag is set to YES (the default) Doxygen
# will generate HTML output.
GENERATE_HTML              = @DOXYFILE_GENERATE_HTML@

# The HTML_OUTPUT tag is used to specify where the HTML docs
# will be put.
HTML_OUTPUT                = "@DOXYFILE_HTML_DIR@"
...

# If the GENERATE_LATEX tag is set to YES Doxygen will
# generate Latex output.
GENERATE_LATEX             = @DOXYFILE_GENERATE_LATEX@

# The LATEX_OUTPUT tag is used to specify where the LaTeX docs
# will be put. If a relative path is entered the value of
# OUTPUT_DIRECTORY will be put in front of it.
LATEX_OUTPUT               = "@DOXYFILE_LATEX_DIR@"

```

When the doxyfile is generated by cmake, all fields surrounded by the @ tag in docyfile.in file are automatically set according to corresponding cmake variables.

INSTALL

The shared and static version of each library will be installed in the adequate folder according to the system requirements. And the folder containing headers of the library will be installed in the include folder of the package framework version.

In CMakeFile.txt of the apps folder:

```

# first application to install
INSTALL(TARGETS <exe name>
  RUNTIME DESTINATION ${INSTALL_BIN_PATH}/${INSTALL_PATH_SUFFIX}

```

```

)
# second application to install : an example
if(<package>_WITH_EXAMPLES)
INSTALL(TARGETS <example name>
  RUNTIME DESTINATION ${INSTALL_BIN_PATH}/${INSTALL_PATH_SUFFIX}
)
endif(<package>_WITH_EXAMPLES)
...

```

Same as for libraries applications will be installed in the bin folder of the package's framework version. If example executables have been built they will also be installed.

In CMakeFile.txt of the `test` folder:

```

if(<package>_WITH_TESTS)
INSTALL(TARGETS <test1 name> <test2 name>
  RUNTIME DESTINATION ${INSTALL_TESTS_PATH}/${INSTALL_PATH_SUFFIX}
)
endif(<package>_WITH_TESTS)

```

Tests will be installed only if they have been built.

In CMakeFile.txt of the `share` folder:

```

if(CMAKE_BUILD_TYPE MATCHES Release)
install(DIRECTORY ${CMAKE_CURRENT_BINARY_DIR}/doc
  DESTINATION ${INSTALL_SHARE_PATH}/doc)
# other documents to install
install(FILE <path to document to install> <path to document to install>
  DESTINATION ${INSTALL_SHARE_PATH})
endif(CMAKE_BUILD_TYPE MATCHES Release)

```

share and **config** folders of the package framework version will receive files. Since these files are the same for debug and release version they will be installed only once, when installing release version.

In CMakeFile.txt of the root folder of the package:

```

if(CMAKE_BUILD_TYPE MATCHES Release)
# mandatory files to install
install(FILE license.txt package.manifest
  DESTINATION ${INSTALL_CONFIG_PATH})

```

Simply copying global files to the package framework version.

6 Good practices

6.1 C/C++ source code writing conventions

This section explains good practices and guidelines for using packages.

6.2 Package development convention

The names of packages are in ascii, lower case, without space or numbers but can include '-' for separating names in compound names.

The names of packages **must be unique** considering all developed packages in the frame of the laboratory.

ICI ajouter tout ce qui concerne le nommage des composants d'À©finis dans les packages.

6.3 Delivery

Packages should be delivered to third parties preferably in binary form, using an OS installer. By delivered we mean:

- an OS installer for the package is created via CMake/CPack process presented before.
- the installer is put on a ftp server.
- the ftp address where to find the installer is referenced in the package.manifest file of the package.
- the package is updated together with the workspace, so that anyone can know the existence of the installer.

The reference installer mechanism is debian packages because ubuntu/debian is the reference OS and debian packaging system is far most powerful, considering dependencies management, than basic archive based packaging. Each time a package is delivered in binary form, a corresponding debian package should be provided, unless impossible (e.g. package is specific to Windows OS).

Each time a new package version is available (tagged in master branch) at least one OS installer (preferably debian package) for that version should be provided.

The policy to reference OS installers in the package.manifest file is the following:

- each time an installer for a new major or minor version of the package is available, it must be added to the package.manifest file.
- each time an installer for a new patch version of the package is available, it must replace the previous patch version for the same minor version.

6.4 Dependencies management

Dependencies are requirements to other software (libraries and applications) that must be installed in the system. These dependencies are defined in the CMakeFile.txt of a package and appear in OS installers provided for that package if they are used at load time or runtime.

For the sack of simplicity for the user, each dependency should preferably be satisfied by installing the corresponding software **using an OS installer** provided by developers of this software. To easily manage indirect system dependencies, the best choice is to use sophisticated installer system, like debian packaging system when working on Linux.

Managing dependencies should be done while keeping following guideline in mind:

- The number of direct external dependencies should be very limited for a given package (0 to 5 max).
- each **package dependency** in CMakeFile.txt must be satisfied by exactly one OS installer, if used at load time or runtime (for a given OS install system, by default debian).
- each **required system dependency** in CMakeFile.txt must be satisfied by exactly one OS installer, if used at load time or runtime (for a given OS install system, by default debian).
- each **optional system dependency** in CMakeFile.txt may be satisfied by exactly one OS installer (for a given OS install system, by default debian), if used at load time or runtime. The best option is to avoid using an optional dependency if there is no impact on performance of the software.
- each **alternative system dependency** in CMakeFile.txt must be satisfied by exactly one OS installer (for a given OS install system, by default debian). This requires the developers/administrator of the package to choose the best alternative, which requires a compromise between several concerns:

- efficiency of the binary code.
- availability of the OS installer for the dependency. The optimal option is to choose a dependency that is satisfied by default by the target system. A less good option is too choose a dependency whose OS installer is provided. A bad option is to choose a system dependency for which no OS installer is provided.
- re-usability of the dependency. It is always a better choice to use a commonly used dependency (e.g. boost) that can be reused across multiple packages, than a very specific dependency.

6.5 Structuring

The basic guideline is to separate software into many packages that are structured according to a strict "depends" hierarchy as described in figure 5. The "depends" relationship simply describes that a package depends on one or more other packages, it practically means that a package requires another package to be installed on the system either for load time, runtime or build time reason.

Most of time, as developers commonly use shared libraries for linking their libraries or executables these dependencies are load time dependencies and so also appear in OS installer of the package. Nevertheless for optimization or other specific reasons, sometimes the "depends" relationship can be only a build-time dependency. These consideration are completely bound to build process and is not the subject of this section which focuses on the usage of package to structure development projects: the "depends" relationship has to be understood as a "functional" requirement.

From a functional point of view a dependency can generally represent one of these two alternatives relationship:

- an "extension" relation : a child package **extends** a base package if it provides some functionalities that specialize/extends those of the base package. This is a typical relationship when a library extends the class hierarchy provided by a library of the parent package, or/and when a more specialized/complete version of an application/component is provided.
- a "use" relation : a child package **uses** a base package if it provides new functionalities built onto those provided by the base package. This is the case when new libraries are using more basic ones

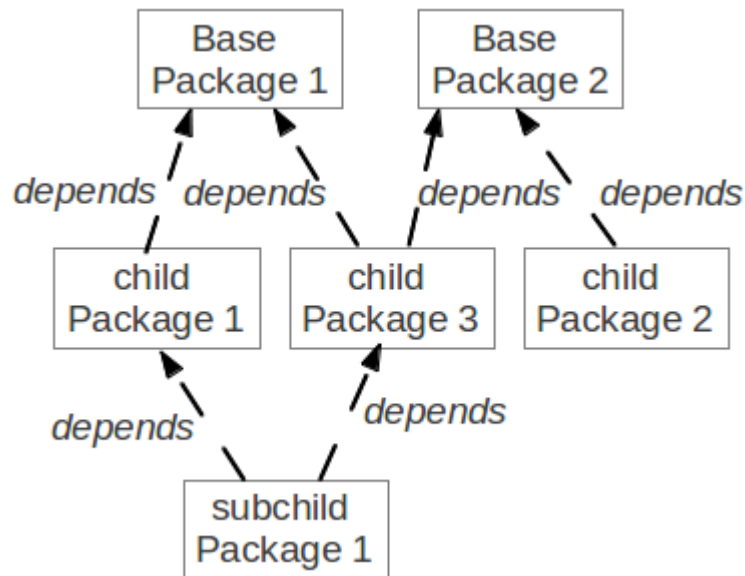


Figure 5: An example hierarchy of packages

or when new applications/component are built using existing applications/components.

Packages Dependencies guideline:

- If a package has some dependencies induced by lower levels of the class hierarchy, prefer making a package for the higher level with less dependencies, and one or more dependent "extension" packages for the lower levels, each of them with the strict minimum required dependencies.
- Prefer an atomic structuring of package by limiting the number of software artefacts generated by a package. Typical use is one package per application or per library (this advice does not targets test cases and examples).
- When developing a component for a given middleware (ROS, OROCOS, etc.) always put the functional code (library) in one package and extend this package each time a component is built from this code.
- Cyclic dependencies are forbidden.
- Dependencies to multiple packages are allowed but the developer should keep in mind to have the lowest possible number of dependencies for a given package.

- Making several packages dependent from a same package is encouraged when these dependencies put in place an "extension" relationship.
- Limit the number of hierarchical level to the strict necessary. As a global guideline we discourage having more than 4 hierarchical levels for the "extension" relationship to keep the extension hierarchy understandable.

Conclusion

Concerning package development, developers have to refer to dedicated document explaining coding rules and general code development guidelines.

Appendix

7 PID cmake functions API

8 Examples

POUBELLE

The workspace also provides categories : each category defines a specific subject that helps classifying packages/frameworks. For instance the category sensor will reference all packages or frameworks that implement the driver of a sensor. Categories can be hierarchically refined into sub categories for instance the category camera refines the category sensors. Categories can be viewed as lists of links that target frameworks/packages. The process of classification of packages is itself coming from package description (i.e. a package specifies the categories it belongs to).

GIT TAGS

Handling tags for version numbers:

- creating the version number (annotated tags):
`git tag -a v1.2.3 -m"<small description of the version>"`
- listing all available versions:
`git tag -l 'v*'`
- showing the information of a given version:
`git show v<version number>`
- getting the released version in package history:
`git checkout tags/v<version number>`

It is also possible to apply tags to certain moment of the package's life in order to register important dates of package state and retrieve this state at any moment in the future. We define two other tag patterns to do so : publication tags, representing software developed to do experiments in the frame of a scientific publication and demo tags used to target versions that match working technical demonstrations. Demo and publication tags relies on last released version number and had specific information:

- **publication tags pattern** have the shape:
`pX.Y[.Z] -<journal-or-conference>-<year>.`
- **demo tags pattern** have the shape:
`dX.Y[.Z] -<name of demo>-<year>.`

GIT BRANCHING

Handling feature branches in GIT:

- Creating a feature branch:
`git checkout -b <feature name> develop`
- Listing available branches:
`git branch -a`
- Incorporating a finished feature on develop:
`git checkout develop`
`git merge -no-ff feature-<feature name>`
`git branch -d feature-<feature name>`
`git push origin develop`

Handling hotfixes branches in GIT:

- Creating a hotfix branch:
`git checkout -b hotfix-<patch version> master`
- Incorporating a finished hotfix on master and develop:
`git checkout master`
`git merge -no-ff hotfix-<patch version>`
`git tag -a <patch version>`
`git push origin master`
`git checkout develop`
`git merge -no-ff hotfix-<patch version>`
`git branch -d hotfix-<patch version>`
`git push origin develop`

COLLABORATIVE WORKING

When an administrator decides to affect a pool of developers to the development of a package:

1. he first clones the official repository of the package into a private access area on the server, that is accessible (via ssh access) only to this pool of developers.
`ssh <private area>`
`git clone <address of the package official repository>`
 2. Second, he creates one or more feature branches as described in previous section.
`git checkout -b <feature name> develop`
 3. Third, all developers in turn clone this private repository into their own work station so that they can work locally and update (push/pull) the server version of the private repository.
`git clone <address of the package private repository>`
 4. When features development is done or aborted, the administrator deletes the private repository on server by simply deleting the repository's folder and removing access to the users if necessary.
`ssh <private area>`
`rm <package private repository folder>`
- Creating a local version of a package repository:
`cd <path to workspace>/packages/`
`git clone <address of the package official repository>`

- Registering a private repository for a given package on administrator workstation:
`cd <path to workspace>/packages/<package>`
`git remote add <private repository name> <private repository address>`
- To get the state of a private repository, without merging the result with administrator's local branches:
`git fetch <private repository name>`
- To remove a reference to a private repository (just after/before deletion of the repository by hand):
`git remote rm <private repository name>`

The development process of a package is always initiated by administrators:

- because they know which packages are already available and so which one can be used or extended in the frame of a master/PHD student or postdoc job.
- because they know who is already working on similar or complementary software.
- because they know which kind of constraints apply to code produced in the laboratory (licenses, access restriction, etc.)

They are so the only persons that can truly:

- orient and help developers to increase their productivity by reusing existing code.
- optimize collaboration between developers.
- centralize the whole process.

the **categories** folder is structured according a set of topics, each topic being itself a folder that in turn contains either other subcategories or **links to frameworks**. **categories** only goal is to help finding available packages (under binary form) according to a given theme. Considering a category, packages that are relevant for the theme it defines are referenced as links (symbolic relative links in Linux) to frameworks folders.

Rules for the modification of the repository:

- `.gitignore` cannot be modified.
- No new direct subfolder or file of the workspace root can be created or removed.
- files (package references, licenses definitions) can be added or removed only by administrators.

8.0.1 Deployment in frameworks

Frameworks are simple repositories in which the binary code or more generally the code usable by third parties is installed. The install process of a package version in its framework is made either :

- by package developers using `make install`, after the compilation phase. The package version is then the current version
- by package users, after downloading the archive of the package, using OS specific primitives (e.g. `dpkg -i`).

In both cases, the `CMakeList.txt` file of a given package defines the way this package is deployed (according to rules defined in this document), either:

- directly, by generating the `make install` procedure.
- indirectly, using `cpack` toolset, by generating the **OS installer** that does the same job, but without having to get access to the package repository.

8.0.2 General policy to manage many package versions

The following rules should be applied whenever developers want to install a given package version:

- Given a package version with given major and minor version numbers, its more recent patch version should always be used. If a new patch version becomes available (e.g. 1.4.2 after a hotfix on 1.4.1) it should replace older patch versions (e.g. 1.4.0 or 1.4.1) if any installed in the package framework. This ensures the developer to use a version with less bug or security problems.

- When a new backward compatible version (e.g. 1.5.0) of a package is to be installed, it may replace the previous version (e.g. 1.4.2) or be simply added to the framework. Most of time it is preferable to test effective compatibility between this new version and the code that was using the old version.
- When a new non backward compatible version of a package is to be installed (e.g. 2.0.0 while 1.4.2 is already installed), it should not immediately replace older versions. The package version is simply added to the package framework, which ensures that other packages can still work with an older version already installed.

When to install ? Generally, there are two main case when developer want to install a package version:

- they want to install software artefacts (libraries, scripts, etc.) coming from a package they are currently developing, in order to use them. In this case, package version is installed in the **own** folder of the framework. This folder is updated each time they compile their package.
- they develop or use a package that requires other packages with version constraints. For a given required package, a released version with number contained between its min and max version constraints has to be installed. In the allowed "interval of versions", the last backward compatible version with last patch version should be installed.

How to install ? There are basically two way to install a given package version:

- from package repository: the developer has to go back to the given version in history (on master branch) and install it.

```
git checkout master
git checkout tags/v<version number>
cd build
{cmake .. - DCMAKE_INSTALL_PREFIX=../../frameworks/
<package name>/<version number>/
make
make install
```
- from an OS installer: it depends on the installer used but the install prefix has to be specified quite the same way as previously.

8.1 Referencing Packages in Workspace

To reference available packages potentially accessible in the workspace, we use the **references** folder in which administrators put **package.manifest** files. In the workspace these files are used to tell to users which packages are available and where they can be found. When a package administrator wants to make the package available to others, he must add the corresponding **package.manifest** in the **references** folder of its workspace and commit the changes to the workspace. All users will then know the package existence whenever they update their workspace. When a package definitively become obsolete, the corresponding manifest file is removed from the **references** folder and the changes into the workspace are committed.

Basic commands

- Referencing a package :
copying the package.manifest file into the references folder of the workspace, and **rename it** in <name of package>.manifest (name of packages are supposed to be unique).
`git add <package name>.manifest`
`git commit -m "<package manifest file> is now referenced"`
`git push origin master`
- Updating available references (when updating the workspace):
`git pull origin master`
- Dereferencing a package
`git rm <package name>.manifest`
`git commit -m "<package manifest file> is no more referenced"`
`git push origin master`

8.2 Licensing packages

The licensing of packages has to be done according to a general politics of the laboratory, team or sub-group. License is let free to their decision and constraints, nevertheless, as software licenses are the same for everybody, licenses definitions are shared between all people.

The **licenses** folder contains definitions for all licenses that are available for use in package description. Each license is defined in a specific file that looks like this for the GPLv3 license (file name is "GNUGPLv3.license") :

```
<license id="GPLv3" fullname="GNU General Public
```

```

License version 3" organization="Free Software Foundation"/>
<description>
/*
* <package_name> : <package_short_description>
* Copyright (C) <year> <authors_name>

* This program is free software: you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation, either version 3 of the License, or
* (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.

* You should have received a copy of the GNU General Public License
* along with this program. If not, see <http://www.gnu.org/licenses/>.
*/
</description>
<legalterms>
//input the full legal terms of the license here
</legalterms>
</license>

```

Data Type Definition:

```

<!DOCTYPE license [

<!ELEMENT license (description, legalterms)>
<!ELEMENT description (CDATA)>
<!ELEMENT legalterms (CDATA)>

<!ATTLIST license id CDATA #REQUIRED>
<!ATTLIST license fullname CDATA #REQUIRED>
<!ATTLIST license organization CDATA #REQUIRED>
]>

```

Explanations:

- the <license> markup is the root of the XML description : it gives the short name of the license used as an identifier (id) in the package

description as well as its complete legal name (**fullname**) and the name of the **organization** that wrote it.

- the **<description>** markup contains a short text that has to be put by developers into any file used in the package (source, interface, test, etc.). Some part have tags with "<" and ">" characters : this is where package specific informations (name, authors, short description etc.) is written by developers.
- the **<legalterms>** markup contains the full legal text of the license. Thanks to this text a license text file can be generated and added with redistribution of packages (e.g. both in repositories and installers).

Important remarks:

- As a package has one license only, using different licenses requires to define different package.
- If a package has one license at a time this license can evolve along time.
- Only administrators can add/remove/modify license files in the workspace.

8.3 Categorizing packages

When developing it is often useful to know which package provide some useful things relative to one or more topics. The aim of the **categories** folder is to standardize the classification and the finding of packages relevant for given concerns.

The **categories** folder is structured into subfolders that describe known categories. Each category folder is itself structured the following way:

- sub folder describes sub categories. A sub category refines the topic of its containing category. For instance folders **arms** and **wheeled vehicles** are contained in the category folder **robot**. These sub categories can in turn be refined the same way.
- symbolic links represent target packages of the category. The name of the link is simply the name of the package, the target of the link is a given package framework. The target itself is always expressed as a **relative path** from the containing category folder to the target framework. By default these links target nothing in the file system as long as the package's framework has not been installed in the **frameworks** folder.

Remark: a same package can be targeted by more than one link: it means that the package belongs to different (sub)categories according to the different point of view of the developers. Doing so, anyone can have his own point of view of the package depending on its center of interest.

To organize the development, defining some "standard categories" should help developers to classify their packages. This requires to list categories of code that is useful for robotic applications development. For instance :

- sensors
 - vision
 - * cameras
 - * kinects
 - * lasers
 - force sensors
 - encoders
- robots
 - arms
 - wheeled vehicles
 - humanoids
- simulators
- navigation
 - cartography
 - positioning
- planning
 - mission
 - path
 - trajectory
- middleware
 - ros
 - contract

– orocos

Of course categories can evolve/change along time, when developers want new categories.

8.3.1 Working with categories

The only people that can modify **categories** folder are package administrators. Each time a new package is created, developers define which are its relevant categories and wrote them in the `package.manifest` file of the package. When an administrator of a package wants to reference it :

- he first copy/rename the `package.manifest` file into the **references** folder.
- according to the relevant categories of the package defined in the manifest, he:
 - creates new (sub)category folders in the the **categories** folder of its workspace, if the corresponding categories do not exist.
 - creates as many symbolic links that target the package framework as there are relevant categories for the package and put these links into the adequate category folders.
- once done he commits its workspace.

From developers point of view, categories just help finding packages that are relevant for given topics. Each time they update their workspace, all available categories and classification of packages according to categories are updated.

To search packages, they can simply navigate in the file system of the **categories** folder.

- if the package is already installed in the **frameworks** folder he can simply follow the link.
- otherwise the link is broken, and the developer has to install the corresponding package.